Cloud Storage encourages users to validate the data they transfer to/from their buckets using either CRC32c or MD5 checksums. This section describes best practices for performing these validations.

There are a variety of ways that data can be corrupted while uploading to or downloading from the Cloud:

- Noisy network links
- Memory errors on client or server computers, or routers along the path
- Software bugs (e.g., in a library that customers use)

To protect against data corruption, Cloud Storage supports two types of hashes: CRC32C and MD5 (described below). Google recommends that customers use CRC32C for all cases, as described in the Validation section below. Customers that prefer MD5 can use that hash, but that hash is not supported for composite objects.

ng: The checksum(s) returned when downloading data from Cloud Storage cover the complete object content. Thus, / a request range there's no way to detect data corruption for that byte range alone. We therefore recommend using :ts only for restarting the download of a full object after the last received offset, because in that case you can compu :e the checksum when the full download completes.

All Cloud Storage objects have a CRC32c hash. CRC32C is a 32-bit Cyclic Redundancy Check (CRC) based on the Castagnoli polynomial. This CRC is described by the IETF specification for SCTP (http://tools.ietf.org/html/rfc4960#appendix-B). Libraries for computing CRC32c include Google's CRC32C (https://github.com/google/crc32c) and Boost (http://www.boost.org/doc/libs/1_68_0/libs/crc/) for C++, crcmod (http://crcmod.sourceforge.net/) for Python, and digest-crc (https://github.com/postmodern/digest-crc) for Ruby. Java users can find an implementation of the algorithm in the GoogleCloudPlatform crc32c Java project (https://github.com/GoogleCloudPlatform/crc32c-java).

The CRC popularly known as CRC32 differs from the CRC32c algorithm used by Cloud Storage.

The Base64 encoded CRC32c is in big-endian byte order.

Cloud Storage supports an MD5 hash for non-composite objects. This hash only applies to a complete object, so it cannot be used to integrity check partial downloads caused by performing a range GET.

Historically, an object's MD5 was expressed via the ETag header. This behavior will continue for non-composite objects in the XML API, but since composite objects don't support MD5 hashes, users should make no assumptions about those ETags except that they will change whenever the underlying data changes, per the specification (https://tools.ietf.org/html/rfc7232#section-2.3).

A download integrity check can be performed by hashing downloaded data on the fly and comparing your results to server-supplied hashes. You should discard downloaded data with incorrect hash values, and you should use retry logic to avoid potentially expensive infinite loops.

Cloud Storage supports server-side validation for uploads, but client-side validation is also a common approach. If your application has already computed the object's MD5 or CRC32c prior to starting the upload, you can supply it with the upload request, and Cloud Storage will only create the object if the hash you provided matches the value we calculated.

Alternatively, users can choose to perform client-side validation by issuing a request for the new object's metadata, comparing the reported hash value, and deleting the object in case of a mismatch. To avoid race conditions where independent processes delete or overwrite each other's data, we also recommend that you use object generations and preconditions (/storage/docs/object-versioning).

In the case of parallel uploads using object composition (/storage/docs/composite-objects), users should perform an integrity check for each component upload and then use component preconditions with their compose requests to protect against race conditions. Object composition

offers no server-side MD5 validation, so users who wish to perform an end-to-end integrity check should apply client-side validation to the new composite object.

At the end of each copy operation, the gsutil (/storage/docs/gsutil) `cp` and `rsync` commands validate that the checksum of the local file matches that of the checksum of the object stored in Cloud Storage. If it does not, gsutil will delete the invalid copy and print a warning message. This very rarely happens. If it does, you can retry the operation.

In the XML API, base64-encoded MD5 and CRC32c hashes are exposed and accepted via the **x-goog-hash header** (/storage/docs/xml-api/reference-headers#xgooghash). In the past, MD5s were used as object ETags, but we recommend that users avoid assuming this since composite objects will use opaque ETag values that make no guarantees outside of changing when the object changes.

Server-side upload validation can be performed by supplying locally computed hashes via the `x-goog-hash` request header. Additionally, the MD5 can be supplied using the standard HTTP Content-MD5 header (/storage/docs/xml-api/reference-headers#contentmd5) (see the specification (https://tools.ietf.org/html/rfc1864#section-2)).

In the JSON API, the objects resource (/storage/docs/json_api/v1/objects) `md5Hash` and `crc32c` properties contain base64-encoded MD5 and CRC32c hashes, respectively. Providing either metadata property is optional. If the properties are not provided in an object insert (/storage/docs/json_api/v1/objects/insert), Cloud Storage calculates the values and writes them to the object's metadata. Supplying either property with an insert request triggers server-side validation for the new object. If Cloud Storage calculates a value for either property that does not match a supplied value, the object is not created. For resumable and multipart uploads, you work with the `md5Hash` and `crc32c` properties in the same way as for a single object insert.